Training Neural Network

Deep Learning (DSE316/616)

Vinod K Kurmi

Assistant Professor, DSE

Indian Institute of Science Education and Research Bhopal

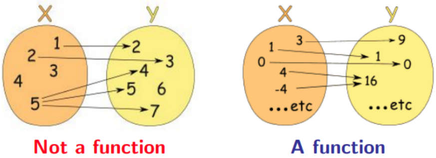
Aug 08, 2022

Disclaimer

- Much of the material and slides for this lecture were borrowed from
  - Bernhard Schölkopf's MLSS 2017 lecture,
  - Tommi Jaakkola's 6.867 class,
  - CMP784: Deep Learning Fall 2021 Erkut Erdem Hacettepe University
  - Fei-Fei Li, Andrej Karpathy and Justin Johnson's CS231n class
  - Hongsheng Li's ELEG5491 class

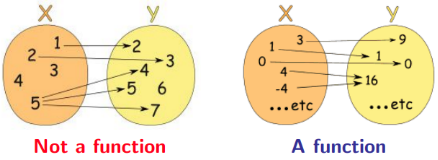
Previous Class: Recap

- Machine learning is '*Function Estimation*' .
- Assumptions: $y = f(x)$.
- Given: $D = \{(x_1^{train}, y_1), (x_2^{train}, y_2), ...(x_n^{train}, y_n)\}$.



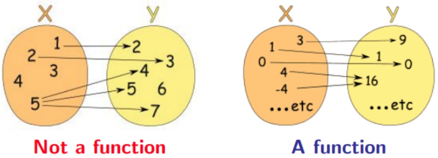
Previous Class: Recap

- Machine learning is '*Function Estimation*' .
- Assumptions: $y = f(x)$.
- Given: $D = \{(x_1^{train}, y_1), (x_2^{train}, y_2), ...(x_n^{train}, y_n)\}$.



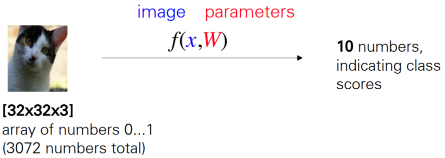
Previous Class: Recap

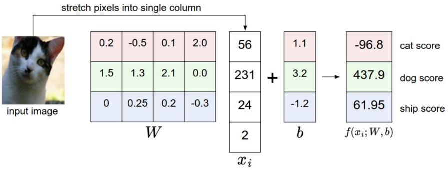
- Machine learning is '*Function Estimation*' .
- Assumptions: $y = f(x)$.
- Given: $D = \{(x_1^{train}, y_1), (x_2^{train}, y_2), ...(x_n^{train}, y_n)\}$.



Previous Class: Recap

- Example of Image classification problem and challenges i.e : viewpoint variation, illumination, occlusion etc.





Previous Class: Recap

Suppose: 3 training examples, 3 classes. With some W the scores are:

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	10.9

Multiclass SVM loss:

- Given an example $\{(x_i, y_i)\}$ where x_i is the image and where $y_i$is the (integer) label, and using the shorthand for the scores vector:$s_i = f(x_i, W)$

the SVM loss has the form:

$$\boxed{\mathcal{L}_i = \sum_{i \neq j} max(0, s_j - s_{y_i} + 1)}$$

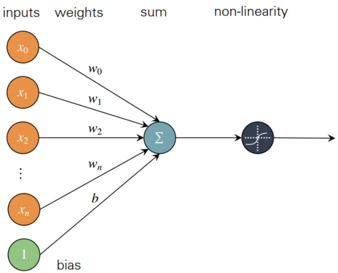
$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i$$

$$\mathbf{L} = (2.9 + 0 + 10.9)/3$$

Previous Class: Recap of Perceptron

- Neuron preactivations (or input activation)

$$a(x) = b + \sum_i w_i x_i = w^T X$$



- Neuron output activation:

$$h(x) = g(a(x)) = g(b + \sum_i w_i x_i)$$

where
w are the weights (parameters)
b is ht bias term
$g(.)$ is the activation function

Previous Class: Recap of Perceptron

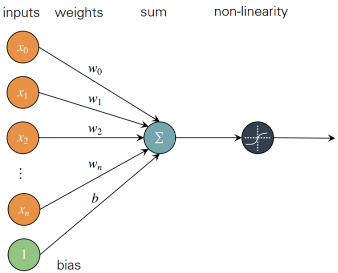
- Neuron preactivations (or input activation)

$$a(x) = b + \sum_i w_i x_i = w^T X$$

- Neuron output activation:

$$h(x) = g(a(x)) = g(b + \sum_i w_i x_i)$$

where
w are the weights (parameters)
b is ht bias term
$g(.)$ is the activation function



Previous Class: Recap of activation functions

- Linear Activation Function
- Sigmoid
- Hyperbolic Tangent (tanh) Activation Function
- ReLu

Multi-Output Perceptron

- Softmax activation function at the output

Previous Class: Recap of activation functions

- Linear Activation Function
- Sigmoid
- Hyperbolic Tangent (tanh) Activation Function
- ReLu

Multi-Output Perceptron

- Softmax activation function at the output

Previous Class: Recap of activation functions

- Linear Activation Function
- Sigmoid
- Hyperbolic Tangent (tanh) Activation Function
- ReLu

Multi-Output Perceptron

- Softmax activation function at the output

Previous Class: Recap of activation functions

- Linear Activation Function
- Sigmoid
- Hyperbolic Tangent (tanh) Activation Function
- ReLu

Multi-Output Perceptron

- Softmax activation function at the output

Previous Class: Recap of activation functions

- Linear Activation Function
- Sigmoid
- Hyperbolic Tangent (tanh) Activation Function
- ReLu

Multi-Output Perceptron

- Softmax activation function at the output

Previous Class:Multi-Layer Perceptron (MLP)

- Consider a network with L hidden layers
  - layer pre-activation for k>1

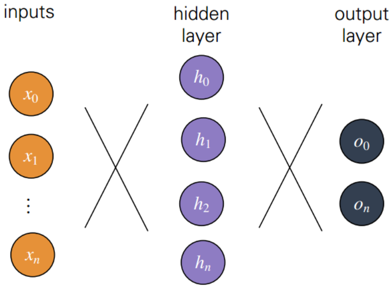
    $$a^{(k)}(x) = b^{(k)} + W^{(k)} h^{(k-1)}(x)$$

  - hidden layer activation from 1 to L

    $$h^{(k)}(x) = g(a^{(k)}(x))$$

  - output layer activation (k=L+1)

    $$h^{(L+1)}(x) = o(a^{(L+1)}(x)) = f(x)$$



$$(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$$

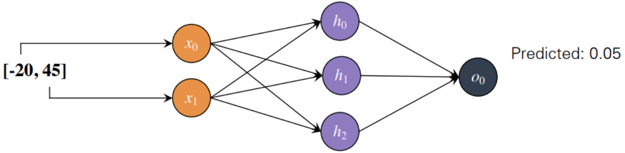
Example Problem: Will my flight be delayed?

- Temperature: -20 F
- Wind Speed: 45 mph

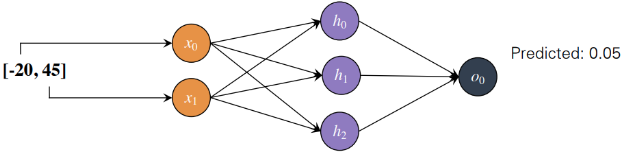
Example Problem: Will my flight be delayed?

- Temperature: -20 F
- Wind Speed: 45 mph

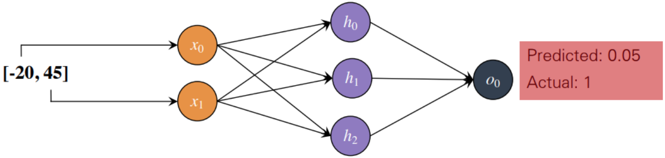
Example Problem: Will my flight be delayed?



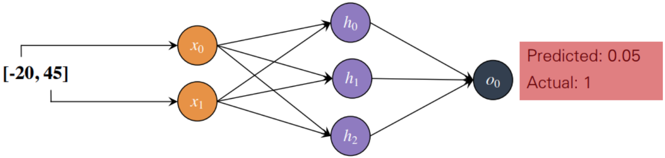
Example Problem: Will my flight be delayed?



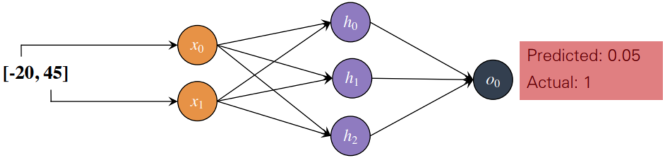
Example Problem: Will my flight be delayed?



Example Problem: Will my flight be delayed?

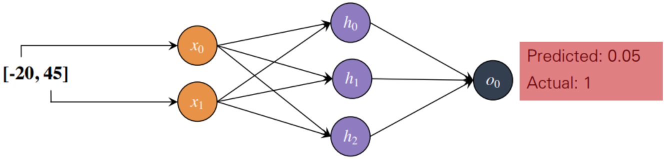


Quantifying Loss



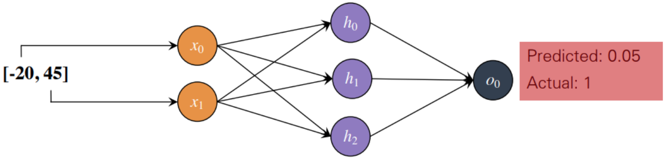
$$\ell(\underbrace{f(\mathbf{x}^{(i)};\theta)}, \underbrace{y^{(i)}}_{\textit{Actual}})$$

Quantifying Loss



$$\ell(\underbrace{f(\mathbf{x}^{(i)};\theta)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Quantifying Loss

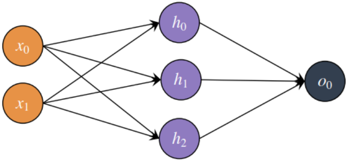


$$\ell(\underbrace{f(\mathbf{x}^{(i)};\theta)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Total Loss

Input

[
[-20, 45],
[80, 0],
[4, 15],
[45, 60],
]



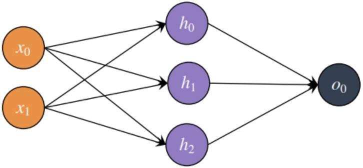
Predicted	Actual
[	[
0.05	1
0.02	0
0.96	1
0.35	1
]	]

$$J(\theta) = \frac{1}{N} \sum_{i} \ell(\underbrace{f(\mathbf{x}^{(i)}; \theta)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Total Loss

Input

[
[-20, 45],
[80, 0],
[4, 15],
[45, 60],
]



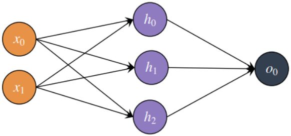
Predicted	Actual
[	[
0.05	1
0.02	0
0.96	1
0.35	1
]	]

$$J(\theta) = \frac{1}{N} \sum_i \ell\big(\underbrace{f(\mathbf{x}^{(i)}; \theta)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\big)$$

Binary Cross Entropy Loss

Input

[
[-20, 45],
[80, 0],
[4, 15],
[45, 60],
]



Predicted	Actual
[	[
0.05	1
0.02	0
0.96	1
0.35	1
]	]

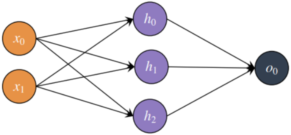
$$\mathcal{J}_{cross-entropy}(\theta) = \frac{1}{N}\sum_i y^{(i)} log(f(\mathbf{x}^{(i)};\theta) + (1 - y^{(i)}) log(1 - f(\mathbf{x}^{(i)};\theta))$$

- For classification problems with a softmax output layer.
- Maximize log-probability of the correct class given an input

Mean Square Error Loss

Input

[
[-20, 45],
[80, 0],
[4, 15],
[45, 60],
]



Predicted	Actual
[	[
0.05	1
0.02	0
0.96	1
0.35	1
]	]

$$\mathcal{J}_{MSE}(\theta) = \frac{1}{N}\sum_{i}(f(\mathbf{x}^{(i)};\theta) - y^{(i)})^2$$

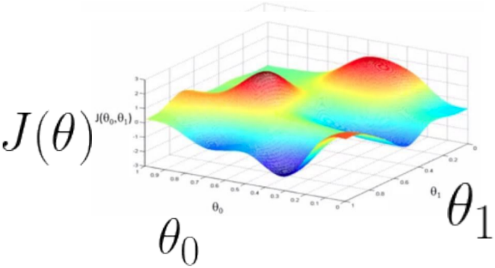
Training

$$\mathcal{J}(\theta) = \frac{1}{N}\sum_i l(\underbrace{f(\mathbf{x}^{(i)};\theta)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

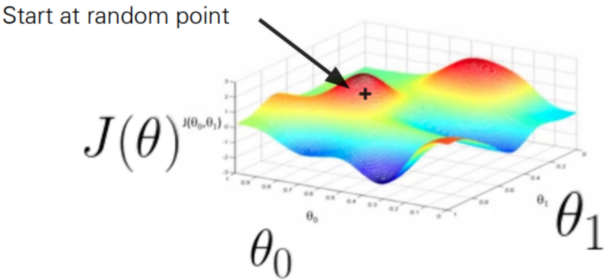
$$\theta = W1, W2, W3...$$

- Learning is cast as optimization.
    - For classification problems, we would like to minimize classification error
    - Loss function can sometimes be viewed as a surrogate for what we want to optimize (e.g. upper bound)

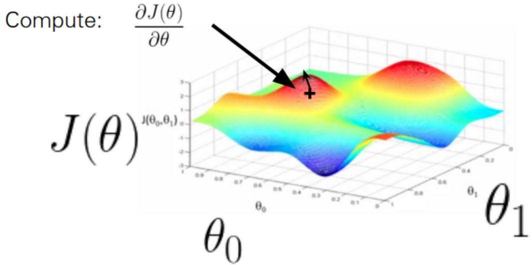
Loss is a function of the model's parameters



How to minimize loss?

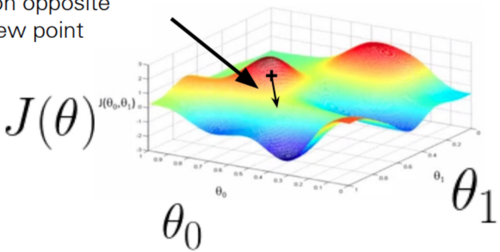


How to minimize loss?

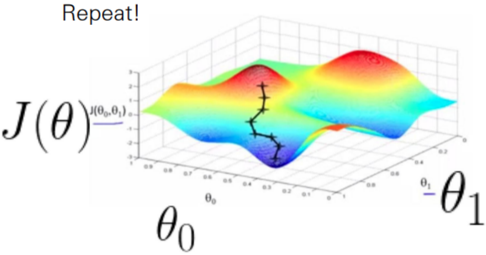


How to minimize loss?

Move in direction opposite
of gradient to new point

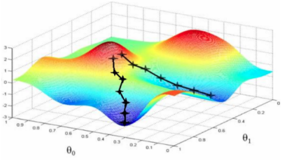


How to minimize loss?



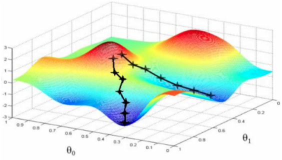
Stochastic Gradient Descent (SGD)

- Initialize θ randomly
- For N Epochs
  - For each training example (x, y):
    - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta}$
    - Update θ with update rule:
      $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$



Gradient Descent (SGD)

- Initialize θ randomly
- For N Epochs
  - For each training example $(\mathbf{x}, y)$:
    - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta}$
    - Update θ with update rule: $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$



Stochastic Gradient Descent?

- Initialize θ randomly
- For N Epochs
  - For each training batch $\{(x_0, y_0), (x_1, y_1), \ldots (x_B, y_B)\}$:
    - Compute Loss Gradient: $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_i \frac{\partial J_i(\theta)}{\partial \theta}$
    - Update θ with update rule: $\theta = \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

Advantages

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates
- Minibatches lead to fast training!
  - Can parallelize computation

Stochastic Gradient Descent?

- Initialize θ randomly
- For N Epochs
    - For each training batch $\{(x_0, y_0), (x_1, y_1), ...(x_B, y_B)\}$:
        - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{1}{B} \sum_i \frac{\partial \mathcal{J}_i(\theta)}{\partial \theta}$
        - Update θ with update rule: $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$

Advantages

- More accurate estimation of gradient
    - Smoother convergence
    - Allows for larger learning rates
- Minibatches lead to fast training!
    - Can parallelize computation

Stochastic Gradient Descent?

- Initialize θ randomly
- For N Epochs
  - For each training batch $\{(x_0, y_0), (x_1, y_1), ...(x_B, y_B)\}$:
    - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{1}{B} \sum_i \frac{\partial \mathcal{J}_i(\theta)}{\partial \theta}$
    - Update θ with update rule: $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$

Advantages

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates
- Minibatches lead to fast training!
  - Can parallelize computation

Stochastic Gradient Descent?

- Initialize θ randomly
- For N Epochs
  - For each training batch $\{(x_0, y_0), (x_1, y_1), ...(x_B, y_B)\}$:
    - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{1}{B} \sum_i \frac{\partial \mathcal{J}_i(\theta)}{\partial \theta}$
    - Update θ with update rule: $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$

Advantages

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates
- Minibatches lead to fast training!
  - Can parallelize computation

Stochastic Gradient Descent?

- Initialize θ randomly
- For N Epochs
  - For each training batch $\{(x_0, y_0), (x_1, y_1), ...(x_B, y_B)\}$:
    - Compute Loss Gradient: $\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{1}{B} \sum_i \frac{\partial \mathcal{J}_i(\theta)}{\partial \theta}$
    - Update θ with update rule: $\theta = \theta - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta}$

Advantages

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates
- Minibatches lead to fast training!
  - Can parallelize computation

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} ... W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} ... W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch $\Delta = \nabla_\theta \ell(f(x^{(i)};\theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)};\theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)};\theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} ... W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch
      $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} \ldots W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch
      $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} \ldots W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch
      $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} \dots W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch
      $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

Stochastic Gradient Descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\theta = \{W^{(1)}, b^{(1)} ... W^{(L+1)}, b^{(L+1)}\}$
  - for N iterations
    - for each training example $(x^{(i)}, y^{(i)})$ or batch
      $\Delta = \nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$
- To apply this algorithm to neural network training, we need
  - the loss function $\ell(f(x^{(i)}; \theta), y^{(i)})$
  - a procedure to compute the parameter gradients $\nabla_\theta \ell(f(x^{(i)}; \theta), y^{(i)})$

What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x;\theta_1,...,\theta_L) = h_L(h_{L-1}(...h_1(x;\theta_1),\theta_{L-1});\theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x;\theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x;\theta_1,...,\theta_L))$$

What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
    - x: input
    - θ_l: parameter of layer l
    - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

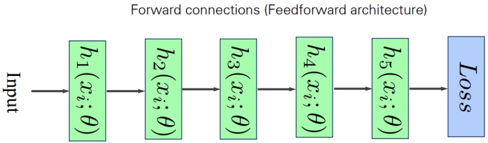
What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions
- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

Neural network models

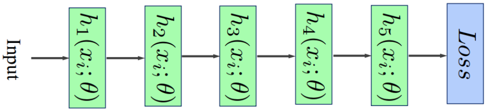
- A neural network model is a series of hierarchically connected functions
- The hierarchy can be very, very complex



Neural network models

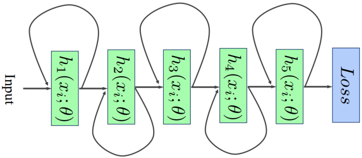
- A neural network model is a series of hierarchically connected functions
- The hierarchy can be very, very complex

Forward connections (Feedforward architecture)



Neural network models

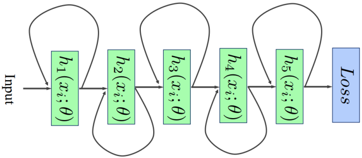
- A neural network model is a series of hierarchically connected functions
- The hierarchy can be very, very complex



Loopy connections (Recurrent architecture, special care needed)

Neural network models

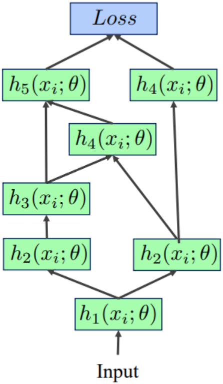
- A neural network model is a series of hierarchically connected functions
- The hierarchy can be very, very complex



Loopy connections (Recurrent architecture, special care needed)

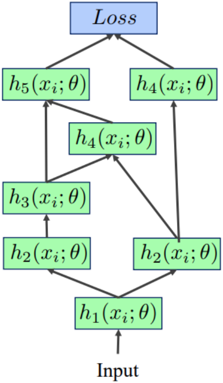
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



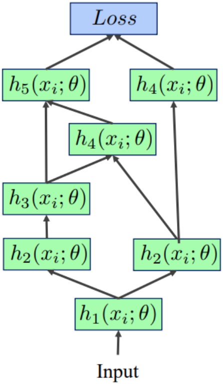
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x;\theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\dots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



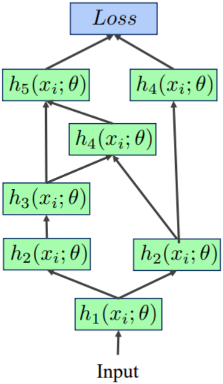
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\dots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



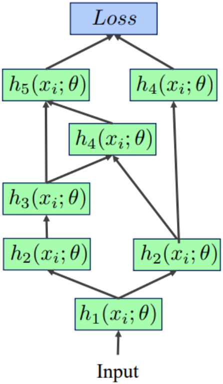
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x;\theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



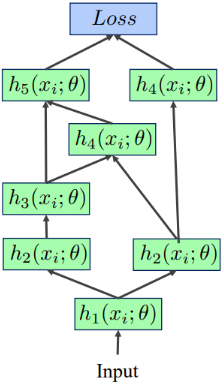
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



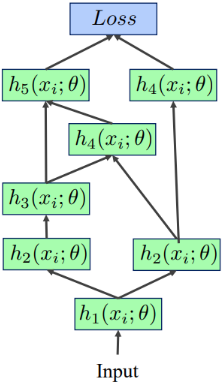
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



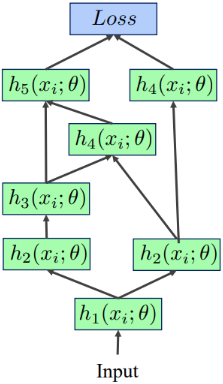
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



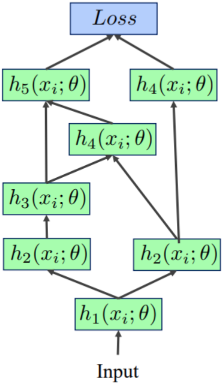
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



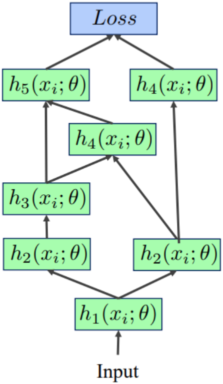
What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



What is a module

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters θ
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation
  - store module input
  - easy to get module output fast
  - easy to compute derivatives



Anything goes or do special constraints exist?

- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a **feedforward** cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later

Anything goes or do special constraints exist?

- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a **feedforward** cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later

Anything goes or do special constraints exist?

- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a **feedforward** cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later

Anything goes or do special constraints exist?

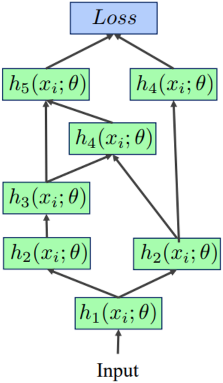
- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a **feedforward** cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later

Anything goes or do special constraints exist?

- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a **feedforward** cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later

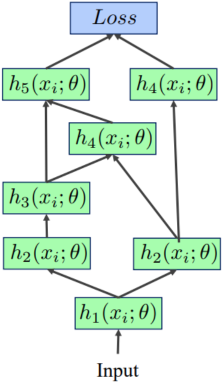
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



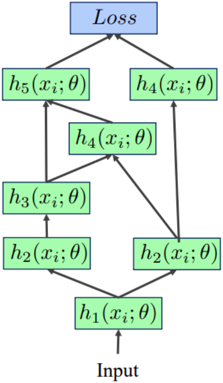
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



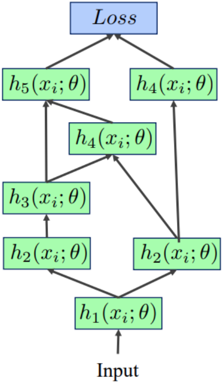
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



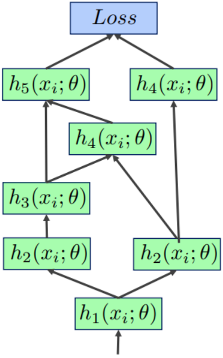
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



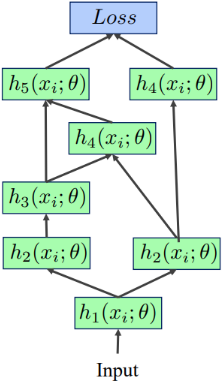
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
    - We need to know the precise function behind each module $h_l(\ldots)$
    - Recursive operations
        - One module's output is another's input
    - Steps
        - Visit modules one by one starting from the data input
        - Some modules might have several inputs from multiple modules
    - Compute modules activations with the **right order**
        - Make sure all the inputs computed at the right time



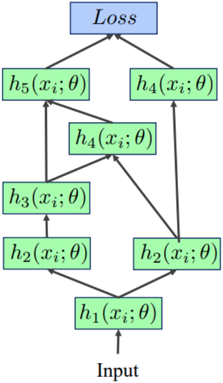
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



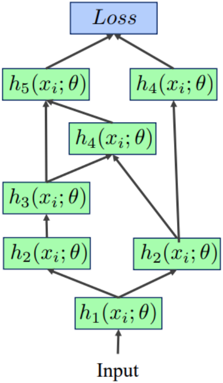
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\dots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



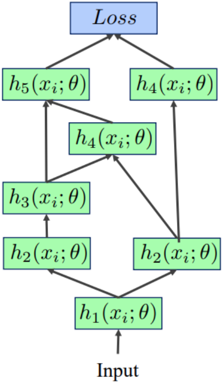
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



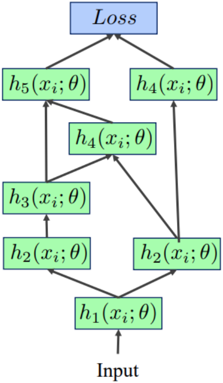
What is a module

- Simply compute the activation of each module in the network $a_l = h_l(x_l; \theta_l)$ were $x_l = a_{l-1}$
  - We need to know the precise function behind each module $h_l(\ldots)$
  - Recursive operations
    - One module's output is another's input
  - Steps
    - Visit modules one by one starting from the data input
    - Some modules might have several inputs from multiple modules
  - Compute modules activations with the **right order**
    - Make sure all the inputs computed at the right time



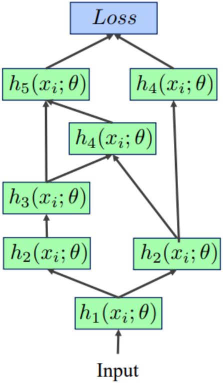
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



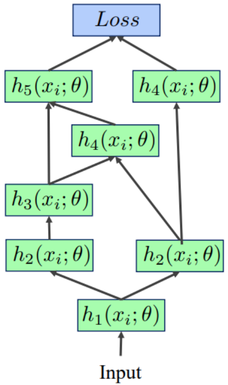
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



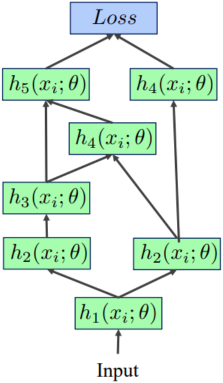
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



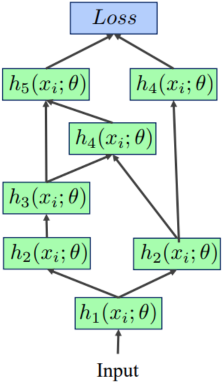
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



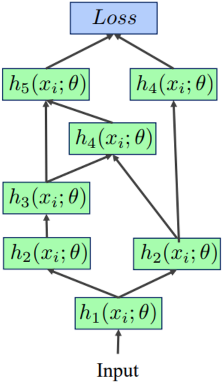
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



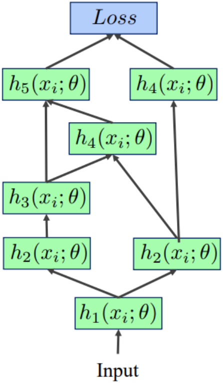
What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



What is a module

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the forward computations first
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



Again, what is a neural network again?

- $a_L(x;\theta_1,...,\theta_L) = h_L(h_{L-1}(...h_1(x;\theta_1),\theta_{L-1});\theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x;\theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x;\theta_1,...,\theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1) = \theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1,2,...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1)=\theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1) = \theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1) = \theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x;\theta_1,...,\theta_L) = h_L(h_{L-1}(...h_1(x;\theta_1),\theta_{L-1});\theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x;\theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x;\theta_1,...,\theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1) = \theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y) \in (X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1) = \theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ... L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x; \theta_1, ..., \theta_L) = h_L(h_{L-1}(...h_1(x; \theta_1), \theta_{L-1}); \theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x; \theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x; \theta_1, ..., \theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1)=\theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

Again, what is a neural network again?

- $a_L(x;\theta_1,...,\theta_L) = h_L(h_{L-1}(...h_1(x;\theta_1),\theta_{L-1});\theta_L)$
  - x: input
  - θ_l: parameter of layer l
  - $a_l = h_l(x;\theta_l)$: (non)-linear function
- Given training corpus X, Y find optimal parameters

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in(X,Y)} \ell(y, a_L(x;\theta_1,...,\theta_L))$$

- To use any gradient descent based optimization

$$\theta^{(t+1)=\theta^t - \eta_t \frac{\partial \mathcal{L}}{\partial \theta_t}}$$

- we need the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$; $l = 1, 2, ...L$
- How to compute the gradients for such a complicated function enclosing other functions, like $a_l(...)$

backpropagation examples and optimizers

Next Class..